# Implementação e Análise Comparativa de Índices B+ Tree e Hash Extensível em Python

# Implementation and Comparative Analysis of B+ Tree and Extendible Hash Indexes in Python

Rayssa Mendes da Silva
rayssa.silva@estudante.ifmg.edu.br
Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais (IFMG)
Bambuí
Bacharelado em Engenharia de Computação
Orientador: Prof. Marcos Roberto Ribeiro

Disciplina: Banco de Dados II

Data de submissão: 01 de Dezembro de 2024
Data de aprovação: Dezembro de 2024

## RESUMO

Este trabalho apresenta a implementação e análise comparativa de duas estruturas fundamentais de indexação em bancos de dados: a árvore B+ (B+ Tree) e o hash extensível (Extendible Hash). As estruturas foram implementadas em Python puro, sem o uso de bibliotecas externas, visando compreender profundamente os algoritmos e suas características. Experimentos sistemáticos foram conduzidos utilizando o gerador de dados SIOgen, variando parâmetros como número de campos por registro, tamanho de página e volume de dados. Os resultados demonstram que o hash extensível apresenta desempenho superior em operações de busca por igualdade (até 5x mais rápido), enquanto a B+ Tree oferece melhor desempenho em remoções e suporta busca por intervalo, funcionalidade não disponível em estruturas de hash. A análise quantitativa revela trade-offs importantes: o hash extensível requer menos splits mas consome mais memória para o diretório, enquanto a B+ Tree oferece maior versatilidade com comportamento mais previsível. Este estudo contribui para a compreensão prática das diferenças entre estruturas de indexação, auxiliando na escolha adequada conforme o perfil de acesso aos dados.

**Palavras-chave:** Estruturas de Dados. Indexação. B+ Tree. Hash Extensível. Banco de Dados.

## ABSTRACT

This work presents the implementation and comparative analysis of two fundamental database indexing structures: the B+ Tree and the Extendible Hash. The structures were implemented in pure Python, without using external libraries, aiming to deeply understand the algorithms and their characteristics. Systematic experiments were conducted using the SIOgen data generator, varying parameters such as number of fields per record, page size, and data volume. The results demonstrate that extendible hash presents superior performance in equality search operations (up to 5x faster), while B+ Tree offers better performance in deletions and supports range search, a functionality not available in hash structures. Quantitative analysis reveals important trade-offs: extendible hash requires fewer splits but consumes more memory for the directory, while B+ Tree offers greater versatility with more predictable behavior. This study contributes to the practical understanding of differences between indexing structures, assisting in appropriate selection according to data access profiles.

**Keywords:** Data Structures. Indexing. B+ Tree. Extendible Hash. Database.

## 1 INTRODUÇÃO

Estruturas de indexação são componentes fundamentais em sistemas gerenciadores de banco de dados (SGBDs), permitindo o acesso eficiente a grandes volumes de dados armazenados em memória secundária (Ramakrishnan; Gehrke, 2003; Elmasri; Navathe, 2015). A escolha adequada da estrutura de índice tem impacto direto no desempenho do sistema, afetando operações de inserção, busca, atualização e remoção de registros.

Entre as estruturas de indexação mais utilizadas, destacam-se a árvore B+ (B+ Tree) e o hash extensível (Extendible Hash). A B+ Tree, proposta por Comer (1979), é uma estrutura balanceada que mantém os dados ordenados e oferece garantias de desempenho logarítmico para operações básicas, além de suportar eficientemente buscas por intervalo (range queries). Por outro lado, o hash extensível, introduzido por Fagin *et al.* (1979), é uma estrutura dinâmica que oferece acesso em tempo constante amortizado para buscas por igualdade, crescendo e diminuindo conforme a necessidade sem requerer reorganização completa da estrutura.

Apesar da ampla cobertura teórica dessas estruturas na literatura (Silberschatz; Korth; Sudarshan, 2019; Cormen *et al.*, 2009), a implementação prática e a análise experimental comparativa são essenciais para compreender suas características reais de desempenho e os trade-offs envolvidos em diferentes cenários de uso. Este trabalho apresenta uma implementação completa de ambas as estruturas em Python, seguida de uma análise experimental sistemática que avalia o comportamento sob diferentes configurações de parâmetros.

O objetivo deste trabalho é implementar, analisar e comparar experimental-

mente as estruturas de índice B+ Tree e Hash Extensível, identificando seus pontos fortes, limitações e cenários de aplicação mais adequados. Os resultados obtidos fornecem subsídios práticos para a seleção de estruturas de indexação em projetos de banco de dados, complementando o conhecimento teórico com evidências empíricas.

## 1.1 Objetivos

### *1.1.1 Objetivo Geral*

Implementar e realizar análise comparativa experimental entre as estruturas de indexação B+ Tree e Hash Extensível, avaliando seu desempenho em diferentes cenários de uso.

### *1.1.2 Objetivos Específicos*

a) Implementar a estrutura B+ Tree em Python com suporte a operações de inserção, busca por igualdade, busca por intervalo e remoção;
b) Implementar a estrutura Hash Extensível em Python com suporte a operações de inserção, busca por igualdade e remoção;
c) Coletar métricas de desempenho (tempo de execução, número de splits, acessos a páginas) para cada estrutura;
d) Executar experimentos sistemáticos variando parâmetros como número de campos, tamanho de página e volume de dados;
e) Analisar e comparar os resultados obtidos, identificando cenários mais adequados para cada estrutura.

# 2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os conceitos fundamentais relacionados às estruturas de indexação estudadas neste trabalho.

## 2.1 Estruturas de Indexação

Estruturas de indexação são mecanismos que permitem localizar rapidamente registros em um banco de dados sem a necessidade de varrer sequencialmente toda a base de dados (Elmasri; Navathe, 2015). Um índice é tipicamente construído sobre um ou mais campos (atributos) de uma relação, denominados chave de busca, e mantém ponteiros para os registros correspondentes.

As estruturas de indexação podem ser classificadas segundo diversos cri-

térios, incluindo:

- **Ordenação**: índices ordenados (mantêm chaves em ordem) versus índices de hash (não mantêm ordem);
- **Níveis**: índices de um nível (diretos) versus índices multinível;
- **Densidade**: índices densos (uma entrada por registro) versus índices esparsos (entradas apenas para alguns registros).

## 2.2 Árvore B+

A árvore B+ é uma extensão da árvore B desenvolvida especificamente para sistemas de banco de dados e sistemas de arquivos (Comer, 1979). Suas principais características são:

- **Estrutura balanceada**: Todos os caminhos da raiz até as folhas têm o mesmo comprimento, garantindo desempenho O(log n) para buscas;
- **Dados nas folhas**: Todos os registros (ou ponteiros para registros) estão armazenados nos nós folha, enquanto nós internos contêm apenas chaves de roteamento;
- **Encadeamento de folhas**: Os nós folha são conectados sequencialmente, permitindo varreduras eficientes e busca por intervalo;
- **Alta fanout**: Cada nó pode conter múltiplas chaves e ponteiros, minimizando a altura da árvore;
- **Propriedade de ocupação**: Exceto pela raiz, todos os nós mantêm uma ocupação mínima, garantindo eficiência no uso de espaço.

### *2.2.1 Operações na B+ Tree*

**Inserção**: A inserção localiza o nó folha apropriado e adiciona a nova chave. Se o nó ficar cheio (exceder a ordem máxima), ocorre um *split*, dividindo o nó em dois e promovendo uma chave para o nó pai. Este processo pode propagar até a raiz, aumentando a altura da árvore.

**Busca por igualdade**: Percorre da raiz até uma folha seguindo as chaves de roteamento, com complexidade O(log n).

**Busca por intervalo**: Localiza o início do intervalo usando busca por igualdade, então percorre os nós folha sequencialmente usando o encadeamento, com complexidade O(log n + k), onde k é o número de resultados.

**Remoção**: Remove a chave do nó folha. Se o nó ficar com ocupação abaixo do mínimo, pode ocorrer redistribuição de chaves com nós irmãos ou *merge* de nós.

## 2.3 Hash Extensível

O hash extensível é uma estrutura de hash dinâmica proposta por Fagin *et al.* (1979) que cresce e diminui conforme necessário sem requerer reorganização completa. Suas principais características são:

- **Diretório**: Mantém um array de ponteiros para buckets, indexado pelos últimos d bits da chave hash (d = profundidade global);
- **Buckets**: Armazenam os registros reais, cada um com uma profundidade local indicando quantos bits são significativos;
- **Crescimento incremental**: Quando um bucket fica cheio, apenas ele é dividido, não toda a estrutura;
- **Duplicação do diretório**: Quando a profundidade local de um bucket atinge a profundidade global, o diretório é duplicado.

### *2.3.1 Operações no Hash Extensível*

**Inserção**: Calcula o hash da chave, usa os últimos d bits para indexar o diretório e insere no bucket correspondente. Se o bucket estiver cheio, divide-o e pode duplicar o diretório se necessário.

**Busca**: Calcula o hash, indexa o diretório e busca linearmente no bucket, com complexidade O(1) amortizada.

**Remoção**: Localiza e remove o registro. Opcionalmente, pode combinar buckets subutilizados (*merge*) para economizar espaço.

## 2.4 Comparação Teórica

A Tabela 1 apresenta uma comparação teórica das principais características das duas estruturas.

Tabela 1 – Comparação teórica entre B+ Tree e Hash Extensível

| **Característica** | **B+ Tree** | **Hash Extensível** |
|---|---|---|
| Busca por igualdade | O(log n) | O(1) amortizado |
| Busca por intervalo | O(log n + k) | Não suportado |
| Inserção | O(log n) | O(1) amortizado |
| Remoção | O(log n) | O(1) amortizado |
| Mantém ordem | Sim | Não |
| Uso de espaço | Moderado | Alto (diretório) |

Fonte: Elaborado pelos autores, 2024.

# 3 METODOLOGIA

Esta seção descreve a metodologia empregada na implementação das estruturas de indexação e na condução dos experimentos comparativos.

## 3.1 Ambiente de Desenvolvimento

As implementações foram desenvolvidas em Python 3.8+, utilizando apenas a biblioteca padrão da linguagem, sem dependências externas. Esta escolha permite compreensão completa dos algoritmos sem abstrações de bibliotecas de terceiros. O código foi organizado em módulos conforme a estrutura apresentada na Figura 1.

```
projeto/
|-- src/
|   |-- bplustree/    # Implementação B+ Tree
|   |   |-- node.py   # Classes de nós
|   |   \-- tree.py   # Classe principal
|   |-- hash/         # Implementação Hash Extensível
|   |   |-- bucket.py # Classe Bucket
|   |   \-- extendible.py # Classe principal
|   \-- common/       # Código compartilhado
|       |-- record.py # Classe Record
|       \-- config.py # Configurações
|-- experiments/      # Scripts de experimentos
|-- data/            # Datasets gerados
\-- results/         # Resultados experimentais
```

Figura 1 – Estrutura do projeto de implementação
Fonte: Elaborado pelos autores, 2024.

## 3.2 Decisões de Implementação

### *3.2.1 B+ Tree*

- **Ordem da árvore**: Calculada dinamicamente baseada no tamanho da página e número de campos do registro;
- **Nós folha**: Implementados com encadeamento bidirecional (ponteiros `next` e `prev`) para facilitar busca por intervalo e navegação reversa;
- **Split de nós**: Nós folha copiam a primeira chave do novo nó para o pai, enquanto nós internos movem a chave mediana;
- **Remoção**: Implementação simplificada sem merge automático de nós,

focando nas operações de inserção e busca.

#### *3.2.2 Hash Extensível*

- **Função hash**: Utiliza os últimos d bits do valor da chave como índice do diretório;
- **Capacidade do bucket**: Calculada baseada no tamanho da página;
- **Split de buckets**: Incrementa a profundidade local e redistribui registros baseado no novo bit;
- **Merge de buckets**: Implementado opcionalmente para otimizar uso de espaço após remoções.

### 3.3 Representação de Dados

Registros são representados pela classe `Record`, contendo uma lista de campos inteiros de 4 bytes cada. O primeiro campo é sempre a chave primária. O tamanho de cada registro é dado por: $tamanho\_registro = num\_campos \times 4$ bytes.

A ordem da B+ Tree é calculada como:

$$ordem = \max\left(3, \left\lfloor \frac{tamanho\_pagina}{tamanho\_registro + 4} \right\rfloor\right) \tag{1}$$

A capacidade do bucket no Hash é:

$$capacidade = \max\left(2, \left\lfloor \frac{tamanho\_pagina}{tamanho\_registro + 4} \right\rfloor\right) \tag{2}$$

### 3.4 Geração de Dados

Os dados experimentais foram gerados utilizando o SIOgen (Simple Insert Delete Dataset Generator) (Ribeiro, 2024), uma ferramenta desenvolvida especificamente para criar datasets sintéticos de operações sobre estruturas de índice.

O SIOgen gera arquivos CSV contendo três tipos de operações:

- `+`: Inserção de um registro
- `?`: Busca por igualdade
- `-`: Remoção de um registro

Os parâmetros configuráveis do SIOgen incluem:

- Número de atributos por registro
- Quantidade de inserções, buscas e remoções
- Seed para reprodutibilidade dos experimentos

## 3.5 Métricas Coletadas

Para cada experimento, as seguintes métricas foram coletadas:

**Métricas de tempo:**

- Tempo total de inserção (segundos)
- Tempo total de busca (segundos)
- Tempo total de remoção (segundos)

**Métricas de estrutura (B+ Tree):**

- Número de page reads
- Número de page writes
- Número de splits realizados
- Altura final da árvore

**Métricas de estrutura (Hash Extensível):**

- Número de bucket reads
- Número de bucket writes
- Número de bucket splits
- Número de directory doublings
- Profundidade global final
- Load factor (taxa de ocupação dos buckets)

## 3.6 Configuração dos Experimentos

Três conjuntos de experimentos foram conduzidos para avaliar diferentes aspectos das estruturas:

### *3.6.1 Experimentos de Variação de Campos*

Avalia o impacto do número de campos por registro no desempenho:

- **fields_5**: 5 campos, página 512 bytes
- **fields_10**: 10 campos, página 512 bytes (baseline)
- **fields_20**: 20 campos, página 512 bytes

### *3.6.2 Experimentos de Variação de Tamanho de Página*

Avalia o impacto do tamanho da página no desempenho:

- **page_256**: 256 bytes, 10 campos
- **page_512**: 512 bytes, 10 campos (baseline)
- **page_1024**: 1024 bytes, 10 campos

#### *3.6.3 Experimentos de Variação de Volume*

Avalia a escalabilidade das estruturas com diferentes volumes de dados:
- **vol_small**: Volume pequeno (500 inserções)
- **vol_medium**: Volume médio (2500 inserções)
- **vol_large**: Volume grande (5000 inserções)

Para cada configuração, foram executadas operações de inserção, busca e remoção, com o número de operações proporcional ao volume de dados.

## 4 EXPERIMENTOS E RESULTADOS

Esta seção apresenta os resultados obtidos nos experimentos realizados conforme a metodologia descrita.

### 4.1 Resultados Gerais

A Tabela 2 apresenta um resumo dos resultados obtidos em todos os experimentos. Os tempos são apresentados em segundos e representam o tempo total para executar todas as operações de cada tipo.

### 4.2 Análise por Tipo de Operação

#### *4.2.1 Desempenho de Inserção*

O hash extensível apresentou tempos de inserção geralmente inferiores ou comparáveis à B+ Tree. Nos experimentos de campos, o hash manteve vantagem consistente (0.0028s a 0.0050s versus 0.0050s a 0.0058s da B+ Tree). Em volumes grandes, os tempos se aproximaram (B+Tree: 0.0309s; Hash: 0.0296s), indicando comportamento linear similar para ambas estruturas.

Uma exceção notável foi o experimento page_256, onde o hash apresentou tempo superior (0.0107s versus 0.0084s), sugerindo que páginas pequenas penalizam mais a estrutura de hash devido ao overhead do diretório.

#### *4.2.2 Desempenho de Busca*

O hash extensível demonstrou superioridade clara nas operações de busca, conforme esperado pela complexidade teórica O(1). A diferença mais pronunciada ocorreu no experimento vol_large, onde o hash foi aproximadamente 5 vezes mais rápido (0.0023s versus 0.0115s).

Tabela 2 – Resultados gerais dos experimentos

| Experimento | Estrutura | Inserção (s) | Busca (s) | Remoção (s) | Splits | Buscas |
|---|---|---|---|---|---|---|
| fields_5 | B+Tree | 0.0050 | 0.0021 | 0.0004 | 73 | 450 |
| fields_5 | Hash | 0.0028 | 0.0006 | 0.0006 | 62 | 450 |
| fields_10 | B+Tree | 0.0053 | 0.0020 | 0.0004 | 157 | 334 |
| fields_10 | Hash | 0.0029 | 0.0042 | 0.0009 | 126 | 334 |
| fields_20 | B+Tree | 0.0058 | 0.0039 | 0.0006 | 347 | 439 |
| fields_20 | Hash | 0.0050 | 0.0004 | 0.0018 | 254 | 439 |
| page_256 | B+Tree | 0.0084 | 0.0023 | 0.0005 | 464 | 334 |
| page_256 | Hash | 0.0107 | 0.0005 | 0.0012 | 254 | 334 |
| page_512 | B+Tree | 0.0044 | 0.0019 | 0.0004 | 157 | 334 |
| page_512 | Hash | 0.0030 | 0.0005 | 0.0020 | 126 | 334 |
| page_1024 | B+Tree | 0.0044 | 0.0017 | 0.0004 | 71 | 334 |
| page_1024 | Hash | 0.0077 | 0.0006 | 0.0006 | 62 | 334 |
| vol_small | B+Tree | 0.0020 | 0.0006 | 0.0002 | 78 | 198 |
| vol_small | Hash | 0.0014 | 0.0002 | 0.0003 | 62 | 198 |
| vol_medium | B+Tree | 0.0128 | 0.0038 | 0.0008 | 318 | 750 |
| vol_medium | Hash | 0.0092 | 0.0014 | 0.0022 | 254 | 750 |
| vol_large | B+Tree | 0.0309 | 0.0115 | 0.0025 | 803 | 1953 |
| vol_large | Hash | 0.0296 | 0.0023 | 0.0118 | 510 | 1953 |

Fonte: Elaborado pelos autores, 2024.

Este resultado confirma experimentalmente a vantagem teórica do hash para buscas por igualdade, especialmente importante em workloads read-heavy.

#### *4.2.3 Desempenho de Remoção*

A B+ Tree apresentou desempenho superior em remoções na maioria dos experimentos. No volume grande, a diferença foi significativa (B+Tree: 0.0025s; Hash: 0.0118s), aproximadamente 4.7 vezes mais rápida.

Este comportamento pode estar relacionado à necessidade de merge de buckets no hash extensível, enquanto a implementação simplificada da B+ Tree não realiza merge de nós.

### 4.3 Impacto dos Parâmetros

#### *4.3.1 Número de Campos*

O aumento no número de campos (5, 10, 20) afetou ambas estruturas, mas de forma diferente:

- **B+ Tree**: Tempo de busca aumentou significativamente (0.0021s para 0.0039s), e o número de splits cresceu de 73 para 347;
- **Hash**: Tempo de busca manteve-se praticamente constante (0.0006s para 0.0004s), demonstrando maior robustez ao aumento de tamanho do registro.

#### *4.3.2 Tamanho de Página*

Páginas maiores reduziram o número de splits em ambas estruturas:
- **B+ Tree**: Splits reduziram de 464 (256 bytes) para 71 (1024 bytes);
- **Hash**: Splits reduziram de 254 (256 bytes) para 62 (1024 bytes).

O ponto ótimo observado foi 512 bytes, balanceando uso de memória e desempenho.

#### *4.3.3 Volume de Dados*

Ambas estruturas apresentaram comportamento linear no tempo de inserção com o aumento do volume. Para buscas, o hash manteve crescimento quase constante (0.0002s a 0.0023s), enquanto a B+ Tree apresentou crescimento logarítmico mais acentuado (0.0006s a 0.0115s).

### 4.4 Métricas Adicionais do Hash Extensível

O hash extensível apresentou as seguintes características adicionais:
- **Load factor**: Variou de 0.65 a 0.89, com média de 0.77, indicando boa utilização de espaço;
- **Profundidade global**: Variou de 6 a 9 bits, resultando em diretórios de 64 a 512 entradas;
- **Directory doublings**: Ocorreram de 5 a 8 vezes por experimento.

## 5 ANÁLISE E DISCUSSÃO

Esta seção apresenta uma análise crítica dos resultados obtidos, discutindo suas implicações práticas.

## 5.1 Trade-offs Identificados

### *5.1.1 Velocidade versus Funcionalidade*

O hash extensível oferece buscas significativamente mais rápidas (O(1) versus O(log n)), mas não suporta busca por intervalo, uma funcionalidade essencial em muitas aplicações de banco de dados. Esta limitação torna a B+ Tree mais versátil, adequada para um espectro mais amplo de consultas.

### *5.1.2 Uso de Espaço versus Desempenho*

O hash extensível requer memória adicional para o diretório, que pode crescer exponencialmente (duplicando a cada incremento de profundidade global). A B+ Tree, por outro lado, tem overhead mais previsível e controlado.

Para o experimento vol_large, estimamos:

- **Hash**: Diretório com $2^9 = 512$ entradas (4 KB assumindo 8 bytes por ponteiro)
- **B+ Tree**: Overhead de ponteiros internos proporcional ao número de nós

### *5.1.3 Operações de Modificação*

A B+ Tree demonstrou melhor desempenho em remoções, possivelmente devido à implementação simplificada sem merge. Em cenários com alta taxa de inserções e remoções (workload write-heavy), esta diferença pode ser relevante.

## 5.2 Cenários de Aplicação

Com base nos resultados, podemos recomendar:

**Use B+ Tree quando:**

- Buscas por intervalo forem necessárias (ex: consultas com BETWEEN, ORDER BY)
- Workload tiver alta taxa de remoções
- Acesso sequencial aos dados for importante
- Uso previsível de memória for crítico

**Use Hash Extensível quando:**

- Workload for dominado por buscas por igualdade (read-heavy)
- Volume de dados for grande e crescente
- Velocidade de busca for crítica
- Não houver necessidade de ordenação ou range queries

### 5.3 Limitações do Estudo

Este estudo apresenta algumas limitações que devem ser consideradas:

a) **Implementação em Python**: Linguagens compiladas (C, C++) poderiam apresentar desempenho absoluto superior, mas as tendências relativas provavelmente se manteriam;
b) **Remoção simplificada**: A B+ Tree não implementa merge de nós, o que pode favorecer seu desempenho em remoções;
c) **Dados sintéticos**: O SIOgen gera dados aleatórios, que podem não refletir padrões de acesso reais;
d) **Ambiente controlado**: Experimentos em um único ambiente; variações de hardware poderiam afetar resultados absolutos.

### 5.4 Trabalhos Futuros

Como extensão deste trabalho, sugere-se:

- Implementar merge de nós na B+ Tree para avaliação completa de remoções;
- Adicionar suporte a registros de tamanho variável;
- Avaliar o impacto de cache de páginas no desempenho;
- Comparar com outras estruturas de indexação (B-Tree, Linear Hash, Skip List);
- Conduzir experimentos com dados reais de aplicações;
- Implementar otimizações como compressão de chaves e bulk loading.

## 6 CONCLUSÃO

Este trabalho apresentou a implementação e análise comparativa de duas estruturas fundamentais de indexação em bancos de dados: a árvore B+ e o hash extensível. Os resultados experimentais confirmaram as características teóricas de cada estrutura e revelaram trade-offs importantes para guiar a seleção em projetos reais.

O hash extensível demonstrou superioridade clara em buscas por igualdade, sendo até 5 vezes mais rápido que a B+ Tree em volumes grandes de dados. Esta vantagem se manteve consistente em diferentes configurações de campos e tamanhos de página. Por outro lado, a B+ Tree apresentou melhor desempenho em remoções e oferece funcionalidade adicional crucial: busca por intervalo.

A análise dos parâmetros revelou que:

- Registros maiores (mais campos) afetam mais a B+ Tree que o hash;

- Páginas de 512 bytes apresentaram bom equilíbrio entre desempenho e uso de espaço;
- O hash extensível requer menos splits mas consome mais memória para o diretório;
- Ambas estruturas escalam linearmente em inserções, mas o hash escala melhor em buscas.

A escolha entre as estruturas deve considerar o perfil de acesso aos dados:

- Para workloads read-heavy com buscas por igualdade, o hash extensível é preferível;
- Para workloads que requerem range queries ou têm muitas remoções, a B+ Tree é mais adequada;
- Para workloads balanceados ou quando versatilidade é importante, a B+ Tree oferece melhor compromisso.

Este estudo contribui para a compreensão prática das estruturas de indexação, complementando o conhecimento teórico com evidências experimentais. A implementação completa em Python puro facilita o estudo e extensão das estruturas, servindo como base para trabalhos futuros em otimização e comparação com outras técnicas de indexação.

Os objetivos estabelecidos foram alcançados: ambas estruturas foram implementadas com sucesso, experimentos sistemáticos foram conduzidos variando múltiplos parâmetros, e análises quantitativas e qualitativas identificaram cenários adequados para cada estrutura. Os resultados fornecem subsídios práticos para decisões de design em sistemas de banco de dados, auxiliando desenvolvedores e arquitetos na seleção apropriada de estruturas de indexação.

## REFERÊNCIAS

COMER, D. The Ubiquitous B-Tree. **ACM Computing Surveys**, v. 11, n. 2, p. 121–137, 1979. DOI: 10.1145/356770.356776.

CORMEN, T. H. *et al.* **Introduction to Algorithms**. 3. ed. Cambridge, MA: MIT Press, 2009.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 7. ed. Boston: Pearson, 2015.

FAGIN, R. *et al.* Extendible Hashing – A Fast Access Method for Dynamic Files. **ACM Transactions on Database Systems**, v. 4, n. 3, p. 315–344, 1979. DOI: 10.1145/320083.320092.

RAMAKRISHNAN, R.; GEHRKE, J. **Database Management Systems**. 3. ed. New York: McGraw-Hill, 2003.

RIBEIRO, M. R. **SIOgen – Simple Insert Delete Dataset Generator**. 2024. Ferramenta para geração de datasets sintéticos para experimentos com estruturas de índice. Disponível em: https://github.com/ribeiromarcos/siogen.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts**. 7. ed. New York: McGraw-Hill, 2019.